Telepaartie

Tassilo Tanneberger

November 20, 2019

Ideen des Algorithmus

Finden der LLL

Eine LLL kann nur entstehen, wenn davor zwei Spalten die gleiche Anzahl Objekte beinhalten(a,b,c ... Spalten) $a = b$. Somit würde auch gleich einen Schritt zuvor gelten z.B.

$$((2a = b \land a < c) \oplus (2b = a \land b < c)) \oplus ((3a = b) \oplus (3b = a)) \tag{1}$$

Mit diesem Wissen können wir mögliche LLLs zwei Schichten voraussagen. Zudem möchte ich noch kurz darauf aufmerksam machen, dass $a > c$ oder $b > c$ bei $\leq$ würde Regel (1) greifen. Diese aufgestellten Beobachtungen helfen uns immer, die zu untersuchende Möglichkeit um zwei Schichten zu reduzieren.

Darstellung als Baum

Die einzig wirklich sinnvolle datenstrukturelle Darstellung ist als Baum, der folgende Eigenschaften besitzt:

- max_c... Maximale Anzahl Kinder und Kanten die ein Knoten hat.
- max_k... Maximale Anzahl an Knoten.
- N... Anzahl der Schichten bis zur ersten LLL

$$max_k = \sum_{i=1}^{N-2} (3i) + 1 \tag{2}$$

Das sind natürlich nur Worst-Case-Betrachtungen und mit intelligenten Aussortierungsvefahren, z.B dem Ausschließen von Doppelten, lässt sich die Gesamtanzahl der Knoten stark minimieren.

Wie schon im ersten Abschnitt angesprochen, können wir die Anzahl der Schichten um zwei reduzieren, weil wir schon nach den Kanten suchen die diese Eigenschaften (1) erfüllen und somit garantiert zur LLL führen.

Suchen

Auch wenn die Datenstruktur "Baum" immer sehr einladend ist rekursiv zu arbeiten (Suchen), ist sie für unseren Fall hier äußerst ungeeignet, da wir damit sehr viele ungenutzte Knoten erzeugen, was sehr unperformant ist. Deswegen lassen wir alle Kinder aller Knoten eine Schicht erzeugen und diese dann abprüfen, ob sie die Kriterien (1) erfüllen. Damit erhält man eine bessere Average Performance.

Rückwärts

$$f(x,y) = \begin{cases} (2x, y-x) & x < y \\ (x-y, 2y) & y < x \end{cases} \tag{3}$$

So können wir die Umkehrfunktion bilden:

$$f^{-1}(x,y) = \begin{cases} (x + \frac{y}{2}, \frac{y}{2}) & x > y \wedge 0 \equiv y(mod 2) \\ (\frac{x}{2}, y + \frac{x}{2}) & y > x \wedge 0 \equiv x(mod 2) \\ (\frac{x}{2}, y + \frac{x}{2}) & x > y \wedge 0 \equiv x(mod 2) \\ (x + \frac{y}{2}, \frac{y}{2}) & y > x \wedge 0 \equiv y(mod 2) \end{cases} \tag{4}$$

Es fällt auf, dass die Funktion f(x, y) nicht bijektiv ist. Hier ein Beispiel dafür: $f^{-1}(f(3,5)) = (1,7) \text{or} (3,5)$ Weil mehrere Kriterien der Funktion gleichzeitig erfüllt sein können, gibt $f^{-1}(x,y)$ nur ein Tupel zurück, aber in der Implementierung würde darauf geachtet werden.

Finden des Schwierigsten

Ich habe dabei ein ähnliches Lösungsverfahren verwendet wie bei der ersten Teilaufgabe mit einem Baum, wobei ich zuerst alle Tubel gebildet habe:

N . . . Number of Items

$$\forall n \in \mathbb{N} \wedge 1 \leq N - n * 2, (n, n, N - n * 2)_n \tag{5}$$

Das sind die Wurzeln des Baumes. Nun nehmen wir uns f^{-1} (5) zur Hand und berechnen damit alle Äste (Kinder). Wobei Kinder nicht aufgenommen werden, die schon tiefer im Baum existieren. Deswegen gibt es eine Suchfunktion.

Der Baum-Generierungs-Prozess wird abgebrochen, wenn einer der äußeren Äste keine Kinder hat ($|childs| = .0$) weil alle schon tiefer im Baum vorkommen. Dieser Knoten ist dann der Knoten, der zurückgegeben wird und mit dem aktuell Besten verglichen wird.

1 Implementierung

Das Programm wurde mit C++ (C11) implementiert. Das hat den Vorteil, dass ich meine Knoten als Objekte und den Parent und die Kinder einfach als Pointer speichern kann. Zudem habe ich den großen Performance-Vorteil, als kompilierte Sprache, den ich definitiv brauche.

Zum Laufen bringen

Ich würde sehr dringend raten, den gcc (g++) → die GNU Build Essentials zu nutzen, weil ich nicht dafür garantieren kann, dass es z.B mit VSCODE oder Clang geht. Die Cmake beinhaltet zwei Targets, wobei Telepaartie_1 die erste Teilaufgabe ist und Telepaartie_2 die zweite. Ich habe mich für 2 Programme entschieden, um der GNU Philosophie zu folgen.

```
$ mkdir build && cd build

$ cmake ..

$ make

$ ./Telepaartie_1 2 4 7

$./Telepaartie_2 63
```

Der Baum

Die Knoten meines Programmes haben sind in zwei Klassen aufgeteilt. Die Klasse TreeNodeForward für die erste Teilaufgabe und TreeNodeBackwards für die zweite Teilaufgabe. Ich hatte versucht mit einer Interfaceklasse ein wenig Ordnung alles ein wenig zu ordnen hatte aber eher dazu geführt das ich nun mehr Redundaten code habe. Zudem wurde das Auslesen schlecter Knoten bei der zweiten Teilaufgabe mit der Zeit immer Komplexer weswegen es z.B auch funktionen wie ‘search‘ oder ‘advancedMove‘ notwenig wurden.

Beispiel Runs

In der Ordner resources sind Screenshots von exemplarischen Runs.

Weitere Optimierungen

Dadurch das wir wirklich überprüfen was das beste ist mit der Distance Funtkion könnten wir alle in diesem Baum vorkommenden Knoten die unter unserem Input vorkommen aus der Liste der zu überprüfenden rausstreichen.

- K ... Menge aller vorkommen Knoten in einer Überprüfung (in dem erstelltem Baum)
- O ... Menge aller noch zu überprüfenden Werte

$$O \setminus K \tag{6}$$

Das hätte den Effekt das die Liste der Elemente wo wir noch die Distance ermitteln müssten sehr schnell kleiner wird.